Cpt S 122 – Data Structures

Custom Templatized Data Structures in C++

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Topics

- Introduction
- Self Referential Classes
- Dynamic Memory Allocation and Data Structures
- Linked List
  - `insert, delete, isEmpty, printList`
- Stacks
  - `push, pop`
- Queues
  - `enqueue, dequeue`
- Trees
  - `insertNode, inOrder, preOrder, postOrder`

Introduction

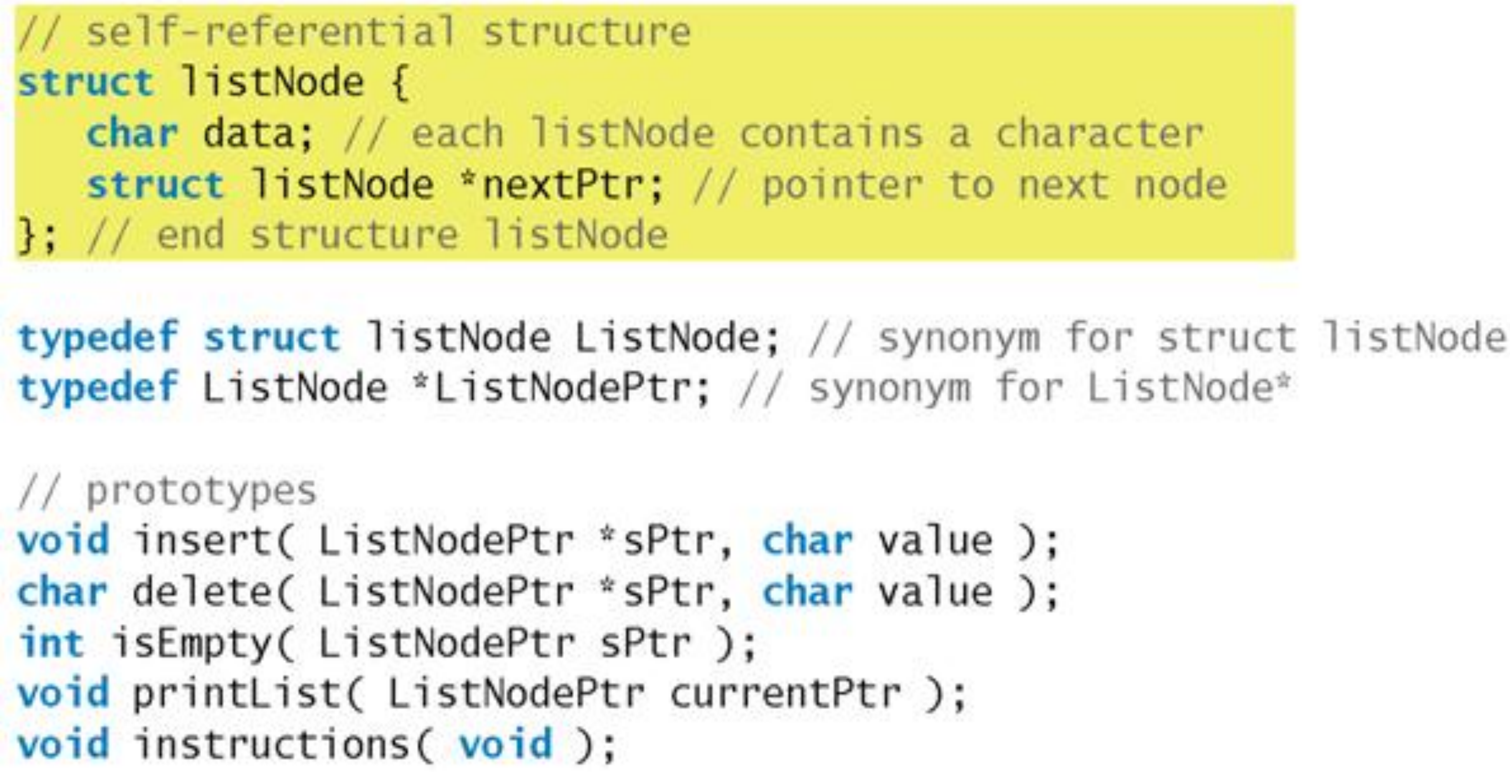
- Fixed-size data structures such as one-dimensional arrays and two-dimensional arrays.
- Dynamic data structures that grow and shrink during execution.
- Linked lists are collections of data items logically "lined up in a row"
  - insertions and removals are made anywhere in a linked list.
- Stacks are important in compilers and operating systems:
  - Insertions and removals are made only at one end of a stack—its top.
- Queues represent waiting lines;
  - insertions are made at the back (also referred to as the tail) of a queue
  - removals are made from the front (also referred to as the head) of a queue.
- Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items,
  - representation of file-system directories
  - compilation of expressions into machine language.

Introduction (cont.)

- Classes, class templates, inheritance and composition is used to create and package these data structures for reusability and maintainability.
- Standard Template Library (STL)
  - The STL is a major portion of the C++ Standard Library.
  - The STL provides *containers, iterators* for traversing those containers
    - algorithms for processing the containers' elements.
  - The STL packages data structures into templatized classes.
  - The STL code is carefully written to be portable, efficient and extensible.

Self-Referential Classes

- A self-referential class contains a pointer member that points to a class object of the same class type.



```
// self-referential structure
struct listNode {
   char data; // each listNode contains a character
   struct listNode *nextPtr; // pointer to next node
}; // end structure listNode

typedef struct listNode ListNode; // synonym for struct listNode
typedef ListNode *ListNodePtr; // synonym for ListNode*

// prototypes
void insert( ListNodePtr *sPtr, char value );
char delete( ListNodePtr *sPtr, char value );
int isEmpty( ListNodePtr sPtr );
void printList( ListNodePtr currentPtr );
void instructions( void );
```

Self-Referential Classes

- A self-referential class contains a pointer member that points to a class object of the same class type.
- Sample **`Node`** class definition:

```
class Node
    {
    public:
       Node( int ); // constructor
       void setData( int ); // set data member
       int getData() const; // get data member
       void setNextPtr( Node * ); // set pointer to next Node
       Node *getNextPtr() const; // get pointer to next Node
    private:
       int data; // data stored in this Node
       Node *nextPtr; // pointer to another object of same type
    }; // end class Node
```

Self-Referential Classes (cont.)

- Member `nextPtr` points to an object of type `Node`
  - another object of the same type as the one being declared here, hence the term “self-referential class.”
- Member `nextPtr` is referred to as a link
  - `nextPtr` can “tie” an object of type `Node` to another object of the same type.
- Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees.
  - Two self-referential class objects linked together to form a list.
  - A null (`0`) pointer is placed in the link member of the second self-referential class object to indicate that the link does not point to another object.
  - A null pointer normally indicates the end of a data structure just as the null character (`'\0'`) indicates the end of a string.

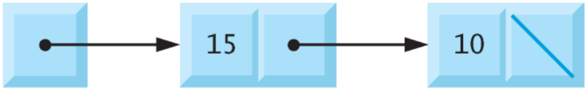


Fig. 20.1 | Two self-referential class objects linked together.

Dynamic Memory Allocation and Data Structures

- The `new` operator takes as an argument
  - the type of the object being dynamically allocated
  - returns a pointer to an object of that type.
- For example, the following statement
  - allocates `sizeof( Node )` bytes,
  - runs the `Node` constructor and assigns the new `Node`'s address to `newPtr`.

    -
      ```
      // create Node with data 10
      Node *newPtr = new Node( 10 );
      ```
- If no memory is available, `new` throws a `bad_alloc` exception.
- The `delete` operator runs the `Node` destructor and deallocates memory allocated with `new`
  - the memory is returned to the system so that the memory can be reallocated in the future.

Linked Lists (cont.)

- If nodes contain base-class pointers to base-class and derived-class objects related by inheritance,
    - we can have a linked list of such nodes and process them polymorphically using `virtual` function calls.
- Stacks and queues are linear data structures
    - can be viewed as constrained versions of linked lists.
- Trees are nonlinear data structures.

Linked Lists Performance

- A linked list is appropriate when the number of data elements to be represented at one time is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Linked lists can be maintained in sorted order
  - By inserting each new element at the proper point in the list.
  - Existing list elements do not need to be moved.
  - Pointers merely need to be updated to point to the correct node.

Linked Lists Performance (cont.)

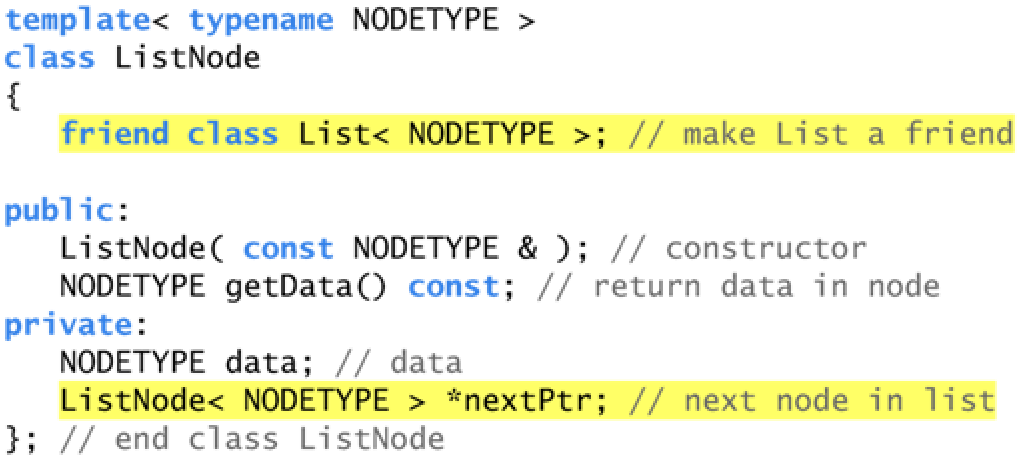
- Insertion & deletion in sorted array is time consuming
    - All the elements following the inserted and deleted elements must be shifted appropriately.
- Linked list allows efficient insertion operations anywhere in the list
- Linked-list nodes are not stored contiguously in memory, but logically they appear to be contiguous.

Linked Lists (cont.)

- The program uses a `List` class template
  - manipulate a list of integer values and a list of floating-point values.
- The program uses class templates
  - `ListNode` and `List`.
- Encapsulated in each `List` object is a linked list of `ListNode` objects.

ListNode Class Template

- Class template `ListNode` contains
  - `private` members `data` and `nextPtr`
  - a constructor to initialize these members and
  - function `getData` to return the data in a node.
- Member `data` stores a value of type `NODETYPE`
  - the type parameter passed to the class template.
- Member `nextPtr` stores a pointer to the next `ListNode` object in the linked list.



```
template< typename NODETYPE >
class ListNode
{
   friend class List< NODETYPE >; // make List a friend

public:
   ListNode( const NODETYPE & ); // constructor
   NODETYPE getData() const; // return data in node
private:
   NODETYPE data; // data
   ListNode< NODETYPE > *nextPtr; // next node in list
}; // end class ListNode
```

Linked Lists (cont.)

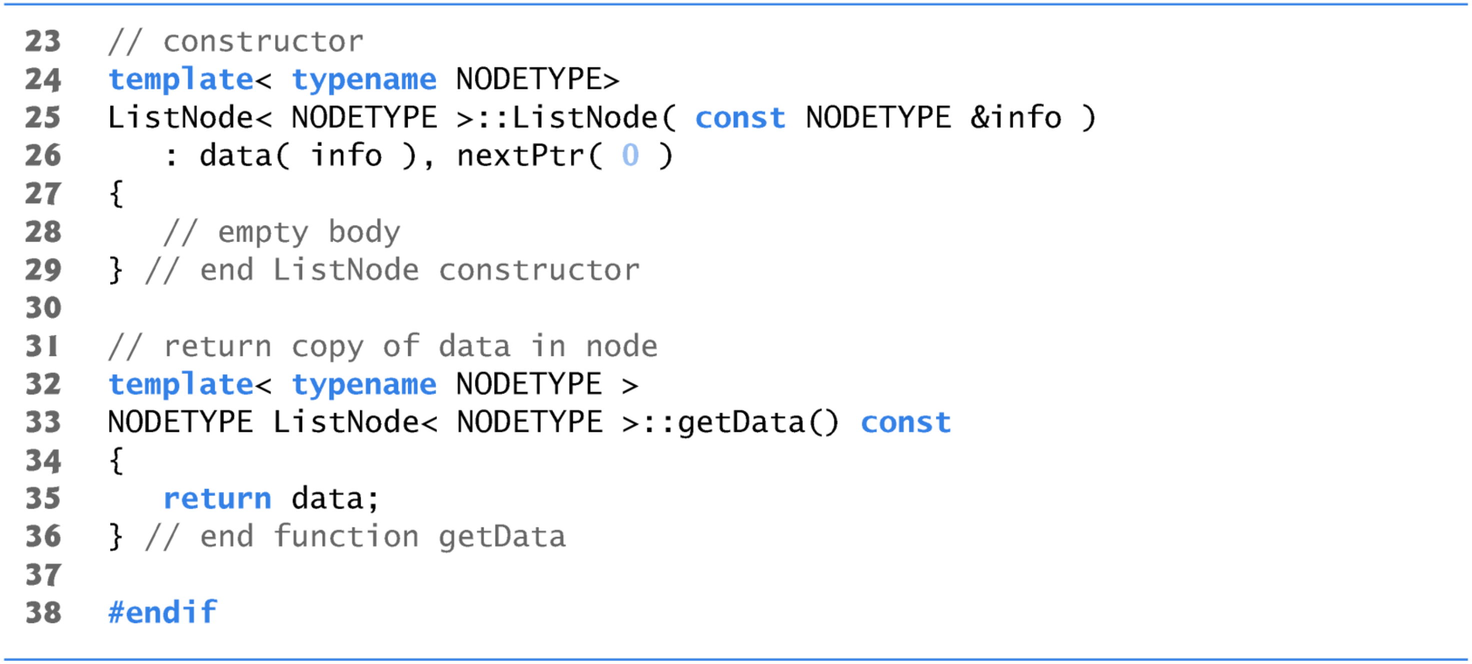
- `ListNode` class template definition declares class `List< NODETYPE >` as a `friend`.
- This makes all member functions of a given specialization of class template `List` friends of the corresponding specialization of class template `ListNode`,
  - so they can access the `private` members of `ListNode` objects of that type.
- `ListNode` template parameter `NODETYPE` is used as the template argument for `List` in the `friend` declaration,
  - `ListNode` specialized with a particular type can be processed only by a `List` specialized with the same type
    - a `List` of `int` values manages `ListNode` objects that store `int` values.

ListNode Template Class

```cpp
 1  // Fig. 20.3: ListNode.h
 2  // Template ListNode class definition.
 3  #ifndef LISTNODE_H
 4  #define LISTNODE_H
 5
 6  // forward declaration of class List required to announce that class
 7  // List exists so it can be used in the friend declaration at line 13
 8  template< typename NODETYPE > class List;
 9
10  template< typename NODETYPE >
11  class ListNode
12  {
13     friend class List< NODETYPE >; // make List a friend
14
15  public:
16     ListNode( const NODETYPE & ); // constructor
17     NODETYPE getData() const; // return data in node
18  private:
19     NODETYPE data; // data
20     ListNode< NODETYPE > *nextPtr; // next node in list
21  }; // end class ListNode
22
```

Fig. 20.3 | `ListNode` class-template definition. (Part 1 of 2.)

ListNode Member Function



```
// constructor
template< typename NODETYPE>
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
   : data( info ), nextPtr( 0 )
{
   // empty body
} // end ListNode constructor

// return copy of data in node
template< typename NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const
{
   return data;
} // end function getData

#endif
```

Fig. 20.3 | `ListNode` class-template definition. (Part 2 of 2.)

List Class Template

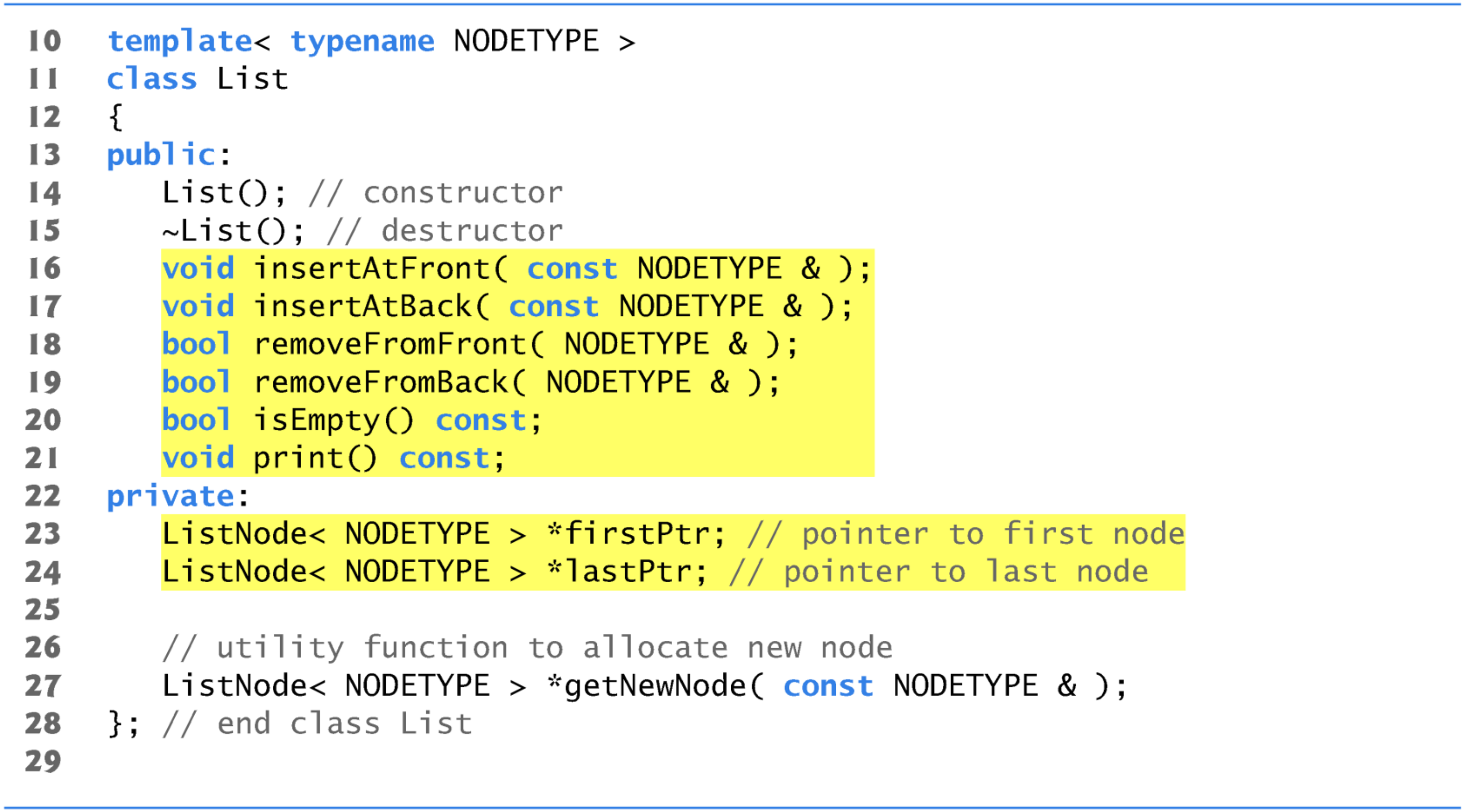
```
// Fig. 20.4: List.h
// Template List class definition.
#ifndef LIST_H
#define LIST_H

#include <iostream>
#include "ListNode.h" // ListNode class definition
using namespace std;

```

Fig. 20.4 | `List` class-template definition. (Part 1 of 10.)

List Class Template



```
template< typename NODETYPE >
class List
{
public:
   List(); // constructor
   ~List(); // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );
   bool isEmpty() const;
   void print() const;
private:
   ListNode< NODETYPE > *firstPtr; // pointer to first node
   ListNode< NODETYPE > *lastPtr; // pointer to last node

   // utility function to allocate new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
}; // end class List

```

Fig. 20.4 | `List` class-template definition. (Part 2 of 10.)

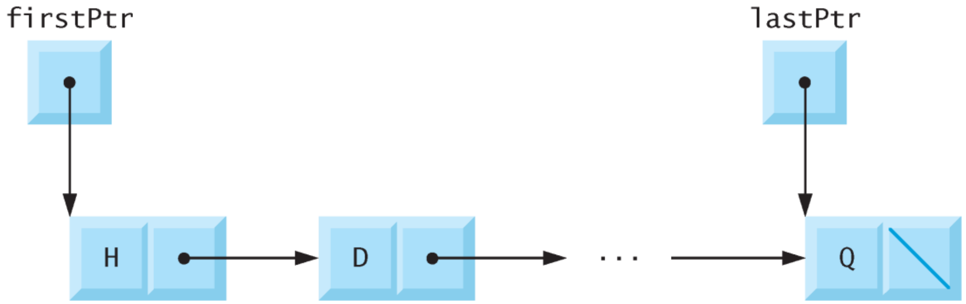


Fig. 20.2 | A graphical representation of a list.

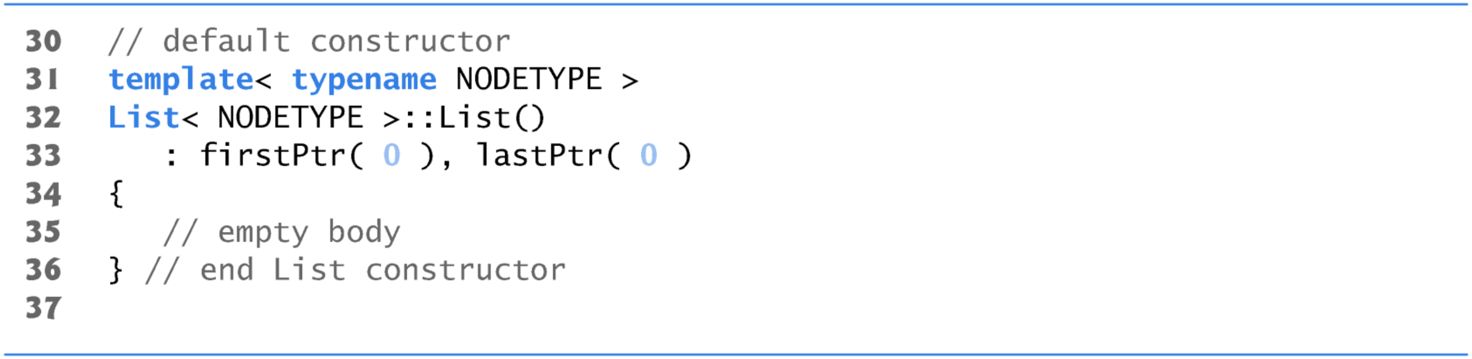
List (cont.)

- `List` class template declare `private` data members
  - `firstPtr` (a pointer to the first `ListNode` in a `List`)
  - `lastPtr` (a pointer to the last `ListNode` in a `List`).
- The default constructor initializes both pointers to `0` (null).
- The destructor ensures that all `ListNode` objects in a `List` object are destroyed when that `List` object is destroyed.

List (cont.)

- The primary `List` functions are
  - `insertAtFront`,
  - `insertAtBack`,
  - `removeFromFront` and
  - `removeFromBack` .
- Function `isEmpty` is called a predicate function
- Function `print` displays the `List`’s contents.
- Utility function `getNewNode` returns a dynamically allocated `ListNode` object.
  - Called from functions `insertAtFront` and `insertAtBack`.

List Class Constructor



```
// default constructor
template< typename NODETYPE >
List< NODETYPE >::List()
   : firstPtr( 0 ), lastPtr( 0 )
{
   // empty body
} // end List constructor

```

Fig. 20.4 | `List` class-template definition. (Part 3 of 10.)

List Class Destructor

```
// destructor
template< typename NODETYPE >
List< NODETYPE >::~List()
{
   if ( !isEmpty() ) // List is not empty
   {
      cout << "Destroying nodes ...\n";

      ListNode< NODETYPE > *currentPtr = firstPtr;
      ListNode< NODETYPE > *tempPtr;

      while ( currentPtr != 0 ) // delete remaining nodes
      {
         tempPtr = currentPtr;
         cout << tempPtr->data << '\n';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
      } // end while
   } // end if

   cout << "All nodes destroyed\n\n";
} // end List destructor
```

Fig. 20.4 | `List` class-template definition. (Part 4 of 10.)

insertAtFront()

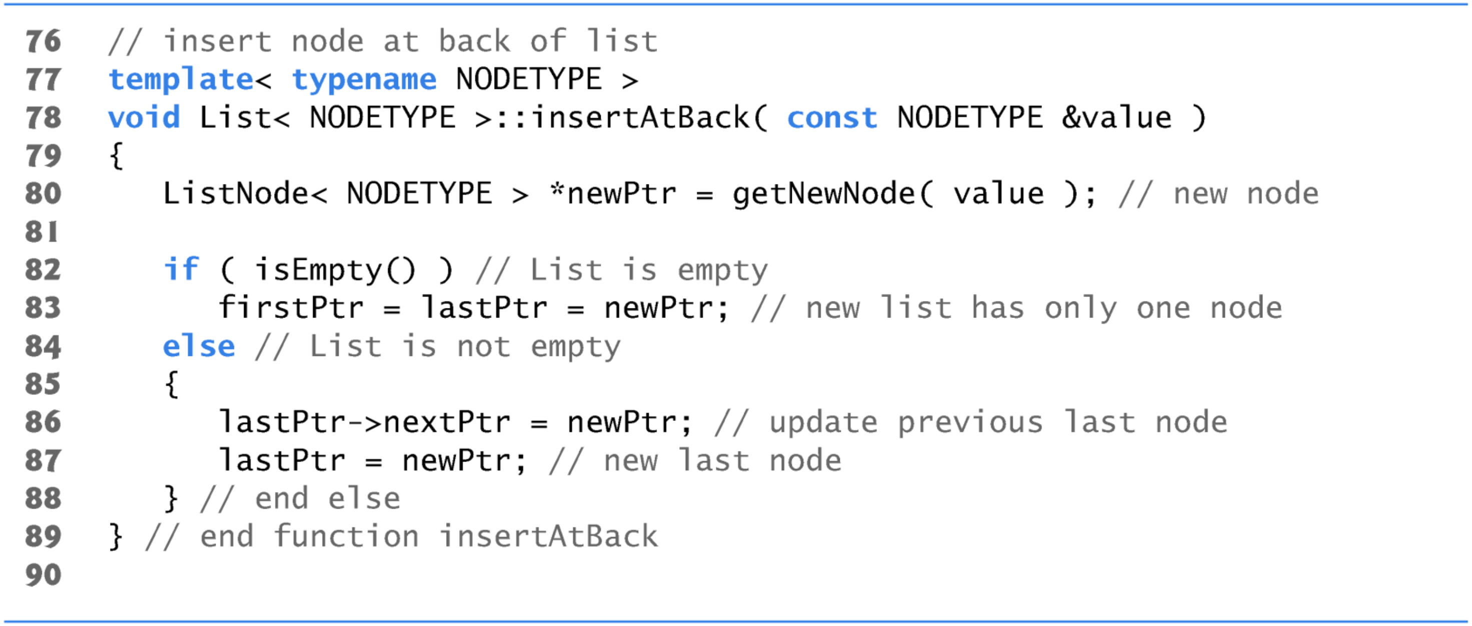
```
// insert node at front of list
template< typename NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
   {
      newPtr->nextPtr = firstPtr; // point new node to previous 1st node
      firstPtr = newPtr; // aim firstPtr at new node
   } // end else
} // end function insertAtFront

```

Fig. 20.4 | List class-template definition. (Part 5 of 10.)

insertAtBack()



```
// insert node at back of list
template< typename NODETYPE >
void List< NODETYPE >::insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
   {
      lastPtr->nextPtr = newPtr; // update previous last node
      lastPtr = newPtr; // new last node
   } // end else
} // end function insertAtBack

```

Fig. 20.4 | List class-template definition. (Part 6 of 10.)

removeFromFront()

```
// delete node from front of list
template< typename NODETYPE >
bool List< NODETYPE >::removeFromFront( NODETYPE &value )
{
   if ( isEmpty() ) // List is empty
      return false; // delete unsuccessful
   else
   {
      ListNode< NODETYPE > *tempPtr = firstPtr; // hold tempPtr to delete

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0; // no nodes remain after removal
      else
         firstPtr = firstPtr->nextPtr; // point to previous 2nd node

      value = tempPtr->data; // return data being removed
      delete tempPtr; // reclaim previous front node
      return true; // delete successful
   } // end else
} // end function removeFromFront

```

Fig. 20.4 | List class-template definition. (Part 7 of 10.)

removeFromBack()

```
// delete node from back of list
template< typename NODETYPE >
bool List< NODETYPE >::removeFromBack( NODETYPE &value )
{
   if ( isEmpty() ) // List is empty
      return false; // delete unsuccessful
   else
   {
      ListNode< NODETYPE > *tempPtr = lastPtr; // hold tempPtr to delete

      if ( firstPtr == lastPtr ) // List has one element
         firstPtr = lastPtr = 0; // no nodes remain after removal
      else
      {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         // locate second-to-last element
         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr; // move to next node

         lastPtr = currentPtr; // remove last node
         currentPtr->nextPtr = 0; // this is now the last node
      } // end else

```

Fig. 20.4 | List class-template definition. (Part 8 of 10.)

isEmpty()

```
        value = tempPtr->data; // return value from old last node
        delete tempPtr; // reclaim former last node
        return true; // delete successful
    } // end else
} // end function removeFromBack

// is List empty?
template< typename NODETYPE >
bool List< NODETYPE >::isEmpty() const
{
    return firstPtr == 0;
} // end function isEmpty

// return pointer to newly allocated node
template< typename NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
    const NODETYPE &value )
{
    return new ListNode< NODETYPE >( value );
} // end function getNewNode

```

Fig. 20.4 | `List` class-template definition. (Part 9 of 10.)

print()

```
// display contents of List
template< typename NODETYPE >
void List< NODETYPE >::print() const
{
   if ( isEmpty() ) // List is empty
   {
      cout << "The list is empty\n\n";
      return;
   } // end if

   ListNode< NODETYPE > *currentPtr = firstPtr;

   cout << "The list is: ";

   while ( currentPtr != 0 ) // get element data
   {
      cout << currentPtr->data << ' ';
      currentPtr = currentPtr->nextPtr;
   } // end while

   cout << "\n\n";
} // end function print

#endif
```

Fig. 20.4 | `List` class-template definition. (Part 10 of 10.)

List (cont.)

- Create `List` objects for types `int` and `double`, respectively.
- Invoke the `testList` function template to manipulate objects.

List

```
// Fig. 20.5: Fig20_05.cpp
// List class test program.
#include <iostream>
#include <string>
#include "List.h" // List class definition
using namespace std;

// display program instructions to user
void instructions()
{
   cout << "Enter one of the following:\n"
      << "  1 to insert at beginning of list\n"
      << "  2 to insert at end of list\n"
      << "  3 to delete from beginning of list\n"
      << "  4 to delete from end of list\n"
      << "  5 to end list processing\n";
} // end function instructions

```

Fig. 20.5 | Manipulating a linked list. (Part 1 of 8.)

```
// function to test a List
template< typename T >
void testList( List< T > &listObject, const string &typeName )
{
   cout << "Testing a List of " << typeName << " values\n";
   instructions(); // display instructions

   int choice; // store user choice
   T value; // store input value

   do // perform user-selected actions
   {
      cout << "? ";
      cin >> choice;

      switch ( choice )
      {
         case 1: // insert at beginning
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtFront( value );
            listObject.print();
            break;
```

Fig. 20.5 | Manipulating a linked list. (Part 2 of 8.)

```
         case 2: // insert at end
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtBack( value );
            listObject.print();
            break;
         case 3: // remove from beginning
            if ( listObject.removeFromFront( value ) )
               cout << value << " removed from list\n";

            listObject.print();
            break;
         case 4: // remove from end
            if ( listObject.removeFromBack( value ) )
               cout << value << " removed from list\n";

            listObject.print();
            break;
      } // end switch
   } while ( choice < 5 ); // end do...while

   cout << "End list test\n\n";
} // end function testList

```

Fig. 20.5 | Manipulating a linked list. (Part 3 of 8.)

```
int main()
{
   // test List of int values
   List< int > integerList;
   testList( integerList, "integer" );

   // test List of double values
   List< double > doubleList;
   testList( doubleList, "double" );
} // end main
```

Fig. 20.5 | Manipulating a linked list. (Part 4 of 8.)

```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3
```

Fig. 20.5 | Manipulating a linked list. (Part 5 of 8.)

```
? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test
```

Fig. 20.5 | Manipulating a linked list. (Part 6 of 8.)

```
Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4
```

Fig. 20.5 | Manipulating a linked list. (Part 7 of 8.)

```
? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3


? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

Fig. 20.5 | Manipulating a linked list. (Part 8 of 8.)

Linked Lists (cont.)

- Singly linked list
  - begins with a pointer to the first node
  - each node contains a pointer to the next node “in sequence.”
- This list terminates with a node whose pointer member has the value 0.
- A singly linked list may be traversed in only one direction.
- A circular singly linked list begins with a pointer to the first node
  - each node contains a pointer to the next node.
- The “last node” does not contain a 0 pointer
  - the pointer in the last node points back to the first node, thus closing the “circle.”

Circular Singly Linked List

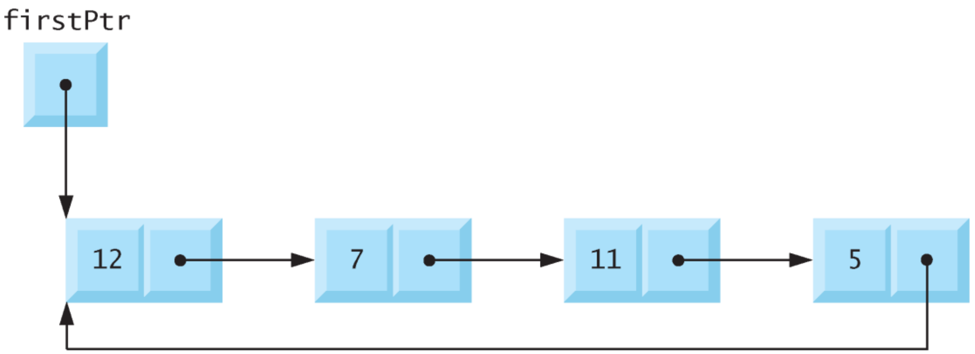


Fig. 20.10 | Circular, singly linked list.

Doubly Linked List

- A doubly linked list allows traversals both forward and backward.
- Implemented with two “start pointers”
  - one that points to the first element of the list to allow front-to-back traversal of the list
  - one that points to the last element to allow back-to-front traversal.
- Each node has both
  - forward pointer to the next node in the list in the forward direction
  - backward pointer to the next node in the list in the backward direction
- List contains an alphabetized telephone directory
  - a search for someone whose name begins with a letter near the front of the alphabet might begin from the front of the list.
  - Searching for someone whose name begins with a letter near the end of the alphabet might begin from the back of the list.

Doubly Linked List

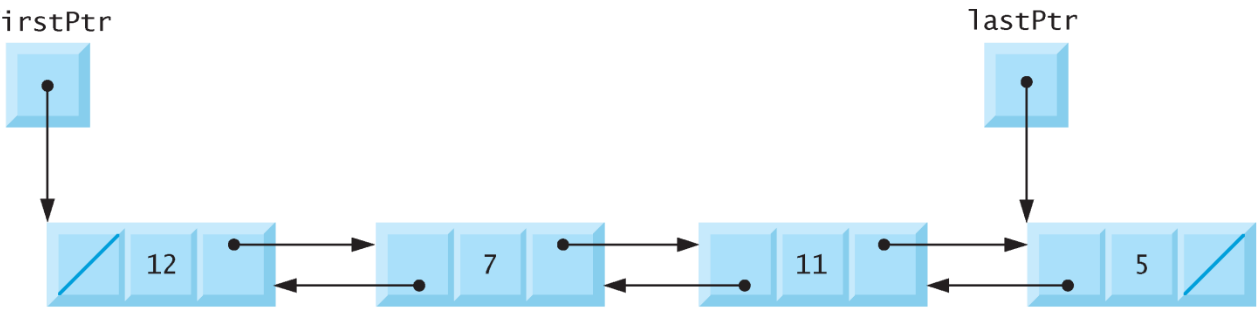


Fig. 20.11 | Doubly linked list.

Circular Doubly Linked List

- Circular doubly linked list
  - forward pointer of the last node points to the first node
  - backward pointer of the first node points to the last node, thus closing the “circle.”

Circular Doubly Linked List

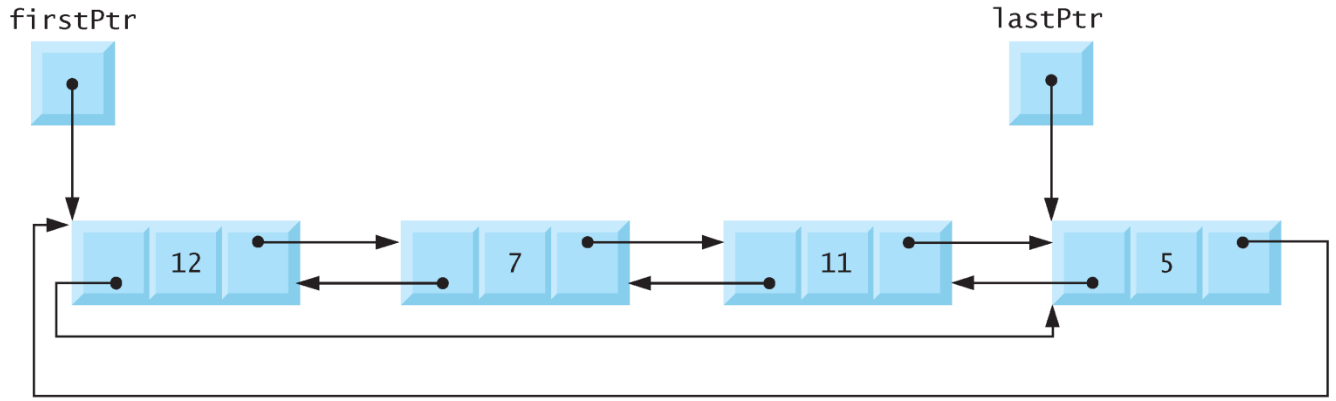


Fig. 20.12 | Circular, doubly linked list.